Automating Administration with Windows PowerShell

Version 24.07

Ahmed Abdelwahed
ahmed@abdelwahed.me
LinkedIn

Getting started with Windows PowerShell

Basic Syntax

- **Basic Command Syntax**: Verb-Noun -Parameter1 value1 -Parameter2 value2
  - **Verb**: Standard action, such as Get, Set, Start, Stop, Add, Remove, etc.
  - **Noun**: The object or resource on which the action is performed, such as Process, Service, Item, etc.
  - **Parameters**: Optional flags or options that modify the behavior of the cmdlet.

General Cmdlets

- **Get-Help**: Retrieves detailed documentation about PowerShell cmdlets, functions, scripts, and usage concepts.
- **Get-Command**: Lists all cmdlets, functions, aliases, and scripts available in the session or retrieves specific commands matching criteria.
- **Get-Member**: Displays the properties and methods of objects output by PowerShell commands, helping to understand object structure and behavior.
- **$PSVersionTable**: Displays details about the PowerShell version and environment.
- **Update-Help**: Downloads and updates help files for PowerShell modules, ensuring the help content is the latest.

File System Operations

- **`Cd ~, mkdir d1, d2, d3:`** Changes the directory to the user's home directory and creates three directories named d1, d2, and d3.
- **`New-Item -Path 'C:\NewFolder' -ItemType Directory: Creates`** a new directory at the specified path.
- **`Copy-Item -Path 'C:\Source\file.txt' -Destination 'C:\Dest': Copies`** a file from one location to another.
- **`Remove-Item -Path 'C:\OldFolder' -Recurse:`** Deletes a folder and all of its contents.

Information Gathering

- **`Get-Command process:`** Finds all commands that include "process" in their name.
- ****`Get-Command Resolve-DnsName |fl ***:`** Lists all properties of the Resolve-DnsName command in full format.
- **`Get-Command -Noun adapter:`** Retrieves commands whose nouns match "adapter".
- **`Get-Command -Verb Set -Noun adapter:`** Retrieves commands with the verb "Set" and noun "adapter".
- **`Get-Command Get-ScheduledTask:`** Retrieves details about the Get-ScheduledTask cmdlet.
- **`Get-ComputerInfo | Get-Member:`** Lists properties and methods of the computer information object.

- **`Get-NetIPConfiguration:`** Displays the IP configuration of the computer.
- **`Get-Volume:`** Displays information about volume(s) on the computer.
- **`Get-Service | Out-String -Stream | Select-String "spooler"`**

Network and System Utilities

- **`Test-Connection:`** Sends ICMP echo request packets ("pings") to a target host and displays the results.
- **`Get-Date; Get-TimeZone:`** Displays the current date and time, and time zone information.
- **`Set-ExecutionPolicy RemoteSigned`**: Changes the PowerShell script execution policy to allow scripts signed by a trusted publisher.

Process Management

- **`Start-Process notepad:`** Starts a new process for Notepad.
- **`Start-Process "cmd" -Verb RunAs:`** Starts the Command Prompt with administrative privileges.
- **`Start-Process "notepad++" -ArgumentList "d:\resources\powershell.txt"`**: Starts Notepad++ with a specific file opened.
- **`Get-Process -Name chrome:`** Retrieves information about processes with the name "chrome".
- **`Get-Process -Name "CCleaner", "Notepad":`** Retrieves processes matching the names "CCleaner" or "Notepad".
- **`Get-LocalUser`**: Retrieves information about local user accounts.
- **`Stop-Process -Name notepad -Force or Stop-Process -Id PID -Force:`** Forcefully stops a process either by name or by process ID.
- **`Get-Process -Name "CCleaner", "Notepad" | Stop-Process -Force:`** Retrieves and forcefully stops processes named "CCleaner" or "Notepad".

Text File Operations

- **`Get-Content -Path 'D:\Resources\PowerShell.txt':`** Displays the contents of the specified text file.
- **`Get-Content -Path 'D:\Resources\PowerShell.txt' -Head 5:`** Displays the first 5 lines of the specified file.
- **`Get-Content -Path 'D:\Resources\PowerShell.txt' -Tail 5`**: Displays the last 5 lines of the specified file.

Working with Process Objects

- **`Get-Process | Select-Object -First 5`**: Selects and displays the first 5 processes.
- **`Get-Process | Select-Object -Last 10:`** Selects and displays the last 10 processes.

PowerShell Module

A PowerShell module is essentially a package of reusable code that can include a variety of functions, cmdlets, and variables, which are grouped together to extend PowerShell capabilities in specific areas. Modules are crucial for sharing code and functionalities among a community or within an organization, making scripting tasks more standardized and efficient.

Commonly Used Module-Related Cmdlets:

- **Get-Module -ListAvailable**: Lists all available modules on the system that can be loaded into the current session. `Get-Module -ListAvailable`
- **$PSHome**: Holds the path to the directory where PowerShell is installed, often where built-in modules and executables are located. `$PSHome`
- **$env:PSModulePath**: Contains the paths where PowerShell looks for modules. This environment variable can be modified to include custom directories. `$env:PSModulePath`
- **Get-Module**: Lists the modules that are currently loaded into the PowerShell session. `Get-Module`
- **Find-Module -Name *active***: Searches online repositories (like the PowerShell Gallery) for modules that match the specified pattern. `Find-Module -Name *active*`
- **Install-Module -Name PowerShellGet**: Installs a module from an online repository, making it available for use. PowerShellGet is a module manager for PowerShell.
  `Install-Module -Name PowerShellGet`
- **Import-Module -Name PowerShellGet**: Loads the specified module into the current session, allowing immediate use of its cmdlets and functionalities. `Import-Module -Name PowerShellGet`
- **Get-Command -Module PowerShellGet**: Lists all commands that are available within a particular module. `Get-Command -Module PowerShellGet`
- **Uninstall-Module -Name <Name> -RequiredVersion <Version>**: Removes a specific version of a module from the system.
  `Uninstall-Module -Name "ModuleName" -RequiredVersion "VersionNumber"`
- **Update-Module -Name <Name>**: Updates a module to the latest version available from the repository.
  `Update-Module -Name "ModuleName"`

Managing PowerShell Outputs

1. **Displaying Output Directly to the Console**: By default, running a command like `Get-Process` sends its output to the console, or stdout.
2. **Redirecting Output to a File**:

- `Get-Process > processes.txt`
- `Get-Process | Out-File -FilePath processes.txt`

3. **Appending Output**:

- `Get-Service >> processes.txt`

4. **Capturing Output in a Variable**: Store command output in a variable for further manipulation:

- `$processes = Get-Process`

5. **Grid View Display**:
    - Output data in an interactive grid view:

- `Get-Process | Out-GridView`

6. **Redirectng Stderr**

- `Get-Process -Name "CCleaner", "Notepad" 2>D:\Resources\error.txt`

7. **Redirecting Stdout and Stderr**:
    - Redirect both stdout and stderr to the same file using `*>`:

- `Get-Process *> outputAndErrors.txt`
- `Get-Process -Name "CCleaner", "Notepad" 2>D:\Resources\error.txt >D:\Resources\error2.txt`

8. **Working with Text Output**:
    - Convert object output to plain text using `Out-String`:

- `$processesText = Get-Process | Out-String`

9. **Redirect output to Printer**

- `Get-Process | Out-Printer`

10. **Mixing Text and Object Output**:
    - Use `Write-Host` for direct text output and standard cmdlets for objects:

- `Write-Host "Computer Host Name is: $hostname"`

Alias in PowerShell

Creating an alias in PowerShell allows you to refer to commands by alternate names, which can be shorter or more memorable than the original command names. You can create aliases for cmdlets, scripts, functions, or executable files.

```
Clear, cls, cp, copy, ls, dir, cd
```

Viewing Existing Aliases

To view a list of all current aliases in your session, use the Get-Alias cmdlet:

- `Get-Alias`
- `Get-Alias -Name list`

Creating a Temporary Alias

- `New-Alias -Name list -Value Get-ChildItem`
- `Set-Alias -Name call -Value Test-NetConnection` #Create a new alias or change an existing one.

This command creates an alias named list for the Get-ChildItem cmdlet, allowing you to use list instead of Get-ChildItem to list items in a directory.

Creating a Persistent Alias

To create an alias that persists across PowerShell sessions, you need to add the alias creation command to your PowerShell profile script:

1. **Open Your PowerShell Profile for Editing:** `notepad $PROFILE`

If the profile doesn't exist, PowerShell will prompt you to create it.

2. **Add the Alias Creation Command to Your Profile:**

- `Set-Alias -Name list -Value Get-ChildItem`
- `Set-Alias -Name call Test-Connection`

Add similar lines for each alias you wish to create. The Set-Alias cmdlet is often used in scripts, including the profile script, because it can overwrite existing aliases.

3. **Save and Close** the profile script. The aliases you added will be created each time you start a PowerShell session.

Create top alias for the top 10 process that consume CPU

```
function Get-TopCpuProcesses {
    Get-Process | Sort-Object -Property CPU -Descending | Select-Object -First
10
}
Set-Alias -Name top -Value Get-TopCpuProcesses
   function top-mem {
    Get-Process | Sort-Object -Property ws -Descending | select -First 10
   }
```

Removing Aliases

To remove an alias within a session, use the Remove-Item cmdlet against the alias:

- `Remove-Item alias:list`

PowerShell Profile Management and Customization

PowerShell profiles allow you to customize your PowerShell environment by automatically executing commands and scripts when you open a new PowerShell session. Here's a breakdown of common profile-related commands and customizations, reformatted for clarity and expanded to include thoughtful descriptions:

1. Viewing and Editing the Profile File

- **Display Profile Details**:
- `$PROFILE | Format-List -Force`
  This command outputs all properties of the $PROFILE variable in a formatted list, helping you understand which profile paths are available and in use.
- **Check for and Create the Profile if it Doesn't Exist**:
- ```
  if (!(Test-Path -Path $PROFILE)) {
      New-Item -ItemType File -Path $PROFILE -Force
  }
  ```
  This snippet checks if the profile file exists and creates one if it does not. This ensures that you have a profile file to customize.
- **Open Profile in Notepad for Editing**:
- `notepad $PROFILE`
  Use this command to open your profile script in Notepad, making it easy to edit and customize commands that run at startup.

2. Profile Inspection and Environment Information

- **Display PowerShell Version Information**:
- `$PSVersionTable`
  This outputs details about your current PowerShell environment, including the version and other key platform characteristics.
- **Echo Profile Path**:
- `echo $PROFILE`
  Simply prints the path to your current PowerShell profile file, which is useful for quick reference.

3. PSReadLine Module and Key Handlers

- **Inspect and Import PSReadLine Module**:
- `Get-Module PSReadLine`
- `Import-Module PSReadLine`
  Check if the PSReadLine module is loaded and import it if needed. PSReadLine enhances the command-line interface with better editing and history capabilities.
- **List and Modify PSReadLine Key Handlers**:
- `Get-PSReadLineKeyHandler | Where-Object { $_.Key -like 'Ctrl+*' }`
- `Get-PSReadLineKeyHandler | Where-Object { $_.Key -like 'Ctrl+*' -and $_.Function -like '*BeginningOfLine*'}`
  This command lists all custom key handlers for Ctrl keys, helping you identify any that might override default behavior.

  ```

- **Setting Custom Key Bindings in PSReadLine**:
- `Set-PSReadLineKeyHandler -Chord Ctrl+q -Function BeginningOfLine`
- `Set-PSReadLineKeyHandler -Chord Ctrl+e -Function EndOfLine`
- `Set-PSReadLineKeyHandler -Chord Ctrl+k -Function KillLine`
- `Set-PSReadLineKeyHandler -Chord Ctrl+u -Function BackwardKillLine`
  These commands set custom key bindings for navigating and modifying lines more efficiently in the PowerShell command line.

4. Custom Functions and Aliases

- **Define a Function to Get Top CPU Consuming Processes**:
- ```
  function top {
   Get-Process | Sort-Object -Property CPU -Descending | Select-Object -First
   10
   }
  ```
  This function retrieves the top 10 processes by CPU usage, useful for quick system diagnostics.

5. Resetting to Default Session

- **Start PowerShell without Any Profiles**:
- `PowerShell -NoProfile`
  This command launches PowerShell without loading any profile scripts, useful for troubleshooting if customizations are causing issues.

  ```

PowerShell Command History Management

- **Viewing Command History**: Use `Get-History` to view the command history for the current session.
- **Running Commands from History**: Retrieve and execute a specific command by ID using `Get-History |Invoke-History -Id 3`.
- **Searching Command History**: To locate a specific previously executed command, use:
  ```
  Get-History | Select-String -Pattern "Get-Process"
  ```
- **Clearing Command History**: `Clear-History` removes all commands from the current session's history without affecting the persistent history file.
- **PowerShell History File**: Commands are stored across sessions in a file located at:
  ```
  C:\Users\<UserName>\AppData\Roaming\Microsoft\Windows\PowerShell\PSReadLine\ConsoleHost_history.txt
  ```
- **Viewing the Entire History File**: Access the full command history by:
  ```
  Get-Content C:\Users\<UserName>\AppData\Roaming\Microsoft\Windows\PowerShell\PSReadLine\ConsoleHost_history.txt
  ```
- **Clearing the Entire History File**: Clear the history file with:
  ```
  Clear-Content -Path "C:\Users\<UserName>\AppData\Roaming\Microsoft\Windows\PowerShell\PSReadLine\ConsoleHost_history.txt"
  ```
- **Configuring History Size**: Set the maximum number of commands to store in history by modifying `$MaximumHistoryCount`:
  ```
  $MaximumHistoryCount = 1000
  ```
  For persistent settings, add `$MaximumHistoryCount` to your `$PROFILE` and edit with:
  ```
  notepad $PROFILE
  ```
- **Saving Both Input and Output**: Record all command inputs and outputs for the session using `Start-Transcript:`
  ```
  Start-Transcript -Path "C:\powershell.txt"
  ```

PowerShell Pipeline Utilization

Objects are passed from one command to another, allowing for efficient processing:

- `Get-ChildItem -Path "C:\Users\Ahmed\Downloads" | Where-Object { $_.Extension -eq '.txt' }`

Selecting, Sorting, and Measuring Objects:

Use `Select-Object, Sort-Object, and Measure-Object` to manipulate and analyze data:

Selecting Properties:

Use Select-Object to select specific properties of objects.

- `New-Item "test1.txt" | Out-Null`
- `Get-Process | Select-Object Name, ID, CPU`
- `Get-Service | select status,name`
- `Get-Service | select -first 1`
- `Get-Service | select -first 2 *`
- `Get-Computerinfo | select CsCaption, CsManufacturer, OsName, CsProcessors, OsTotalVisibleMemorySize, CsDomain`
- `Get-Content log.txt | Select-String  "error|warning|fail"`
- `Select-String "test" -path 'D:\Resources\PowerShell.txt'`
- `Get-Content -Path 'D:\Resources\PowerShell.txt' | Select-String -Pattern "test" -CaseSensitive | Select-Object LineNumber, Line`
- `Get-LocalUser | Select-Object Name, @{Name='AccountStatus'; Expression={if ($_.Enabled) {'Enabled'} else {'Disabled'}}}`

Sorting Objects:

Use Sort-Object to sort objects by one or more properties.

- `Get-Service | Sort-Object Name`
- `Get-Service | Sort -Descending` or `Get-Service | Sort name -Descending`
- `Get-Process | Sort-Object CPU -Descending` #processes consuming the most CPU first
- `Get-Process | Sort-Object -Property ws -Descending | select -First 10` #WS stands for Working Set, which is the current size of the memory occupied by the process.
- `Get-ChildItem -Path "C:\Users\Ahmed\Downloads" | Get-Member -MemberType Properties`
- `Get-ChildItem -Path "C:\Users\Ahmed\Downloads" | Sort Extension`
- `Get-ChildItem -Path "D:\Resources\" -File | Sort-Object -Property Length -Descending`

Measuring Objects:

Use Measure-Object to measure properties of objects, such as count, sum, or average.

- `Get-ChildItem "C:\Users\Ahmed\Downloads" | Measure`
- `Get-ChildItem -Path "C:\Users\Ahmed\Downloads" | Measure-Object -Property Length -Sum -Maximum -Minimum -Average`
- `Get-ChildItem -Path "C:\Users\Ahmed\Downloads" | Measure-Object -Property Length -Sum -Maximum -Minimum -Average | Tee-Object -Variable sizegb`

- `$sizegb.sum / 1mb`
- `$sizegb.sum / 1gb`
- `Get-ChildItem d:\resources | Select-Object Name, @{Name="Size_In_KB";Expression={$_.Length / 1gb}}`
- `Get-ChildItem d:\resources -file | Select-Object Name, @{Name="Size_In_KB";Expression={$_.Length / 1gb}}`
-

Filtering and Enumerating Objects:

Filter and manipulate objects using `Where-Object` and `ForEach-Object`:

- **As a calc**

```
# Addition
        5 + 3  # Outputs 8
# Subtraction
        10 - 2  # Outputs 8
# Multiplication
        4 * 2  # Outputs 8
# Division
        16 / 2  # Outputs 8
#Using Parentheses for Order of Operations
        (2 + 3) * 4  # Outputs 20
```

- `Get-Random`
- `Get-Random -Maximum 100`
- `Get-Random -Minimum -100 -Maximum 100`
- `6 1, 2, 3, 5, 8, 13 | Get-Random`
- `Get-Random -Minimum 1 -Maximum 100 -Count 5`
- # Array of application executables

```
$applications = "notepad", "calc", "cmd"
# Loop through the array and start each application
$applications | ForEach-Object {
  Start-Process $_
} -runas
```

- `Get-Process | Where-Object { $_.CPU -gt 1000 }` # list of processes where the CPU time used is greater than 1000 seconds
- `Get-Process | Where-Object {$_.WorkingSet -gt 1GB}` #This filters processes to only those using more than 1GB of memory.
- `Get-Service | Where-Object { $_.Name -like 'win*' -and $_.Status -eq 'Running' } | Select-Object Name, Status`
- `Get-Service | Where-Object { $_.Name -like 'win*' -and $_.Status -eq 'Running' -or $_.status -eq 'stopped'} | Select-Object Name, Status`

- `Get-Service | Where-Object { ($_.Name -like 'win*' -and $_.Status -eq 'Running') -or ($_.Name -like 'print*' -and $_.Status -eq 'Stopped') } | Select-Object Name, Status`
- `Get-ChildItem -Path "C:\Users\Ahmed\Downloads" | Where-Object { $_.Length -gt 1MB -and $_.Length -lt 5MB }`
- `Get-ChildItem -Path "C:\Users\Ahmed" -Recurse | Where-Object { $_.Extension -eq '.txt' }` #-Recurse gets items in the specified location and all child items.
- `Get-Process -Name Notepad, notepad++ | foreach {$_.kill()}`
- `Get-Process | Where-Object {$_.WS -gt 500MB} | ForEach-Object { $_.Kill() }` **#Terminating Processes by Memory Usage**
- **`Get-Service -Name wuauserv, bits | Where-Object {$_.Status -eq 'Running'} | ForEach-Object { $_.Stop() }`**
- `Get-Process | ForEach-Object { $_.Kill() }` #terminate all running processes

Practical Pipeline Example

- **Find and Export Large Files**:

```
Get-ChildItem -Path C:\ -Recurse |
  Where-Object { $_.Length -gt 500MB } |
  Sort-Object Length -Descending |
  Select-Object Name, Directory, Length |
  Export-Csv -Path C:\largefiles.csv -NoTypeInformation
```

Managing Windows and Network

Windows Initial Configuration

 # Configure network settings (Variables)

```
$interfaceAlias = "Ethernet1" # You may need to adjust this to match your interface alias
$ipAddress = "192.168.244.34"
$subnetMask = 24
$gateway = "192.168.244.2"
$dns = "192.168.244.33"
$newComputerName = "C19"
```

 # Set IP Address

- `New-NetIPAddress -InterfaceAlias $interfaceAlias -IPAddress $ipAddress -PrefixLength $subnetMask -DefaultGateway $gateway`

 # Set DNS Server

- `Set-DnsClientServerAddress -InterfaceAlias $interfaceAlias -ServerAddresses $dns`

 # Enable Remote Desktop

- `Set-ItemProperty -Path 'HKLM:\System\CurrentControlSet\Control\Terminal Server' -name "fDenyTSConnections" -Value 0`

 # Allow RDP through Windows Firewall

- `Enable-NetFirewallRule -DisplayGroup "Remote Desktop"`

 # Start the Remote Desktop Services

- `Set-Service -Name TermService -StartupType Automatic`
- `Start-Service -Name TermService`

 # Disable and Enable Firewall

- `Set-NetFirewallProfile -Profile Domain,Private,Public -Enabled False`
- `Set-NetFirewallProfile -Profile Domain,Private,Public -Enabled True`

#Set the Time Zone

- `Get-TimeZone`
- `Get-TimeZone -ListAvailable`
- `Get-TimeZone -ListAvailable | Where-Object { $_.displayname -like "*muscat" }`
- `Set-TimeZone -Id "Arabian Standard Time"`

 # Change computer name

- `Rename-Computer -NewName $newComputerName -Force -Restart`

#Add computer to domain

- `Add-Computer -domain test.com`

Working With Network

Viewing Network Configuration

Get-NetAdapter: Lists network adapters. Useful for identifying which adapters are available and their status.

- `Get-NetAdapter`
- `Get-NetAdapter | Select Name, InterfaceAlias, InterfaceIndex, LinkSpeed, MacAddress`

Get-NetIPAddress: Retrieves IP address configuration of all network interfaces.

- `Get-NetIPAddress`
- `Get-NetIPAddress -InterfaceAlias "Wi-Fi" -AddressFamily IPv4`
- `Get-NetAdapterStatistics -Name wi-fi | fl *`

Get-NetIPConfiguration: Provides a detailed overview of the network configuration, including IP, DNS, and gateway settings for all network adapters.

- `Get-NetIPConfiguration`
- `Get-NetIPConfiguration -InterfaceIndex 10`

Get-DnsClientServerAddress: Shows the DNS servers assigned to each interface.

- `Test-NetConnection` www.google.com
- `Test-NetConnection -ComputerName 127.0.0.1 -Port 80`
- `Test-NetConnection -ComputerName www.google.com -TraceRoute`
- `pathping www.google.com`
- `Test-Connection yahoo.com`
- `Resolve-DnsName -Name yahoo.com`
- `Resolve-DnsName -Name abdelwahed.me -Type MX`
- `Get-DnsClientDohServerAddress` #retrieve the DNS over HTTPS (DoH) server addresses
- `Get-DnsClientCache`
- `Clear-DnsClientCache`

Modifying Network Configuration

New-NetIPAddress: Assigns a new IP address to a network adapter.

- `New-NetIPAddress -InterfaceAlias "Ethernet1" -IPAddress 192.168.1.100 -PrefixLength 24 -DefaultGateway 192.168.1.1`

Set-NetIPAddress: Modifies the IP address properties of an existing IP address.

- `Set-NetIPAddress -InterfaceAlias "Ethernet1" -IPAddress 192.168.1.100 -PrefixLength 24`

Remove-NetIPAddress: Removes an IP address from a network adapter.

- `Remove-NetIPAddress -InterfaceAlias "Ethernet1" -IPAddress 192.168.1.100`

Set-DnsClientServerAddress: Changes the DNS server addresses for a network adapter.

- `Set-DnsClientServerAddress -InterfaceAlias "wi-fi" -ServerAddresses ("8.8.8.8", "8.8.4.4")`

Resetting Network Configuration

Restart-NetAdapter: Resets a network adapter. This is similar to disabling and re-enabling the adapter in Windows.

- `Restart-NetAdapter -Name "Ethernet1"`

Release and Renewing an IP Address

 # Releasing an IP Address

- `Get-NetIPAddress -InterfaceAlias "Ethernet1" -AddressFamily IPv4 | Remove-NetIPAddress`

 # Disable the network adapter

- `Disable-NetAdapter -Name "Ethernet1"`

 # Enable the network adapter

- `Enable-NetAdapter -Name "Ethernet1"`

ARP and Routing table

 # Viewing the ARP Table in PowerShell

- `Get-NetNeighbor -AddressFamily IPv4`

 # Filtering ARP Entries by Interface

- `Get-NetNeighbor -InterfaceAlias "Ethernet"`

 # Filtering ARP Entries by IP Address

- `Get-NetNeighbor -IPAddress "192.168.1.1"`

 # View All Routing Table Entries

- `Get-NetRoute`

#Add Custom Route

- `New-NetRoute -DestinationPrefix "10.68.33.64/26" -InterfaceAlias "YourInterfaceAlias" -NextHop "10.68.19.193" -PolicyStore PersistentStore`

 # Filter Routes by Destination

- `Get-NetRoute -DestinationPrefix "192.168.1.0/24"`

Port Status

`Test-Port -target 127.0.0.1 -fromPort 80 -toPort 80`

Managing the Windows Firewall

Get-NetFirewallRule: Retrieves firewall rules.

- `Get-NetFirewallRule`

New-NetFirewallRule: Creates a new firewall rule.

- `New-NetFirewallRule -DisplayName "Allow TCP Port 1000" -Direction Inbound -LocalPort 1000 -Protocol TCP -Action Allow`

Set-NetFirewallRule: Modifies existing firewall rules.

- `Set-NetFirewallRule -DisplayName "Existing Rule Name" -NewDisplayName "Updated Rule Name"`

Enable-NetFirewallRule and **Disable-NetFirewallRule**: Enables or disables specified firewall rules.

- `Enable-NetFirewallRule -DisplayName "Allow TCP Port 1000"`

Working with Files and Folders

Listing Files and Folders

- **Get-PSDrive** list partitions`
  ```
  Set-Location -Path C:\Users\ExampleUser\Documents
  ```
- **Get-ChildItem**: Lists the items (files and folders) in a directory.
- ```
  Get-ChildItem -Path C:\Example
  ```
  Use -Recurse to list items in all subdirectories.
- ```
  Get-ChildItem -Hidden
  ```

Creating Directories

- ```
  New-Item -Path 'C:\Example\NewFolder' -ItemType Directory
  ```

Creating Files

- ```
  New-Item -Path 'C:\Example\NewFile.txt' -ItemType File
  "Hello, World!" > C:\Example\NewFile.txt
  ```

Copying Files and Folders

- ```
  Copy-Item -Path 'C:\Example\SourceFile.txt' -Destination
  'C:\Example\DestinationFolder'
  ```

Copying Multiple Files and Folders

- ```
  Copy-Item -Path 'C:\Example\*.txt' -Destination
  'C:\Example\DestinationFolder'
  ```
- ```
  Copy-Item -Path 'C:\Example*' -Destination 'C:\Example\DestinationFolder'
  ```

Moving Files and Folders

- ```
  Move-Item -Path 'C:\Example\OldFolder' -Destination 'C:\Example\NewFolder'
  ```

Deleting Files and Folders

- ```
  Remove-Item -Path 'C:\Example\OldFile.txt'
  ```
  Use -Recurse to delete a folder and its contents.
  ```
  Remove-Item -Path 'C:\Example\OldFile.txt'
  ```

Reading from Files

- ```
  Get-Content -Path 'C:\Example\ReadMe.txt'
  ```

Writing to Files

- **Set-Content:** Writes or replaces content in a file.
  ```
  Set-Content -Path 'd:\resources\powershell.txt' -Value 'New content'
  ```
- **Add-Content:** Appends content to a file.
  ```
  Add-Content -Path "C:\Windows\System32\drivers\etc\hosts" -Value
  175.17.50.20 Master-PC'
  ```

Renaming Files and Folders

- **Rename-Item:** Renames a file or folder.
  ```
  Rename-Item -Path 'C:\Example\OldName.txt' -NewName 'NewName.txt'
  ```

Finding Files and Folders

- **Get-ChildItem** with -Recurse and -Filter or -Include can find files and folders by name or pattern.

```
Get-ChildItem -Path C:\Example -Recurse -Filter "*.txt"
Get-ChildItem -Path d:\resources -Recurse -Filter "*.txt" | Sort-Object
Length -Descending #find all .txt files within the D:\resources directory and its subdirectories, sorting them by size in descending order, meaning the largest files will be listed first.
```

```
Get-ChildItem -Path D:\resources -Recurse -Filter "*.txt" |
    Sort-Object Length -Descending |
    ForEach-Object {
        Copy-Item -Path $_.FullName -Destination D:\folder1}

Get-ChildItem -Path D:\folder1 -Recurse | Select-String -Pattern "test"  |
Select-Object Path, LineNumber
```

Modifying File and Folder Attributes

- **Set-ItemProperty:** Changes properties of files and folders, such as making a file read-only.

```
Set-ItemProperty -Path 'C:\Example\MyFile.txt' -Name IsReadOnly -Value $true
```

Working with File Paths

- **Join-Path:** Combines a path and a child path into a single path.

```
$fullPath = Join-Path -Path 'C:\Example' -ChildPath 'MyFile.txt'
```

Querying management information by using CIM and WMI

Windows Management Instrumentation (WMI) and Common Information Model (CIM) are technologies used in Windows to manage data and operations on computer systems. Both provide a way to interact with system management data from hardware or software components. While WMI uses DCOM (Distributed Component Object Model) as its communication technology, CIM uses WS-Management (a standard web services protocol) which makes it more modern and secure compared to WMI.

Using WMI in PowerShell

WMI can be accessed in PowerShell primarily using the Get-WmiObject cmdlet. This cmdlet allows you to perform queries against WMI classes.

WMI Queries

1. `Get-WmiObject -Namespace root\CIMv2 -List:`
   - Retrieves a list of all management classes available in the root\CIMv2 namespace. This is useful for discovering what management data you can access via WMI.
2. `Get-WmiObject -Namespace root\CIMv2 -List | where name -Like *network*:`
   - Lists all WMI classes in the root\CIMv2 namespace that include "network" in their names. This helps in identifying classes specifically related to network configurations and status.
3. `Get-WmiObject -class *network* -List:`
   - This command seems incorrect because -List with a wildcard class name (-class *network*) generally doesn't yield valid results. Instead, you might use -List to list classes and filter those or drop -List to directly query instances of classes that match the wildcard name.
4. `Get-WmiObject -class Win32_NetworkAdapterConfiguration`:
   - Retrieves instances of the Win32_NetworkAdapterConfiguration class, providing detailed configuration information for each network interface on the system.
5. `Get-WmiObject -Class Win32_OperatingSystem | Get-Member`:
   - Pipes the instances of the Win32_OperatingSystem class to Get-Member, which displays the properties and methods of the class, helping understand what data can be manipulated or retrieved.
6. `Get-WmiObject -Class Win32_OperatingSystem | Select-Object Caption, Version, ServicePackMajorVersion`:
   - Retrieves specific properties of the Win32_OperatingSystem class, specifically the OS name (Caption), version number, and major version number of any installed service pack. This provides a succinct overview of the operating system details.
7. `Get-WmiObject -Class Win32_Processor | Select-Object Name, NumberOfCores, NumberOfLogicalProcessors`:
   - Retrieves and displays selected properties of the Win32_Processor class, showing the processor name, number of cores, and number of logical processors. This is useful for understanding the CPU configuration of the system.

6. `Get-WmiObject -Query "SELECT * FROM Win32_DiskDrive" | Select-Object Model, InterfaceType, Size:`

- Executes a WMI query to retrieve all instances of Win32_DiskDrive and then selects specific properties to display: the model of the disk, the type of interface (e.g., SATA, SCSI), and the size of the disk. This provides detailed physical disk information.

8. **`Get-WmiObject -Class Win32_Service -Filter "State='Running' AND StartMode='Auto'" | Select-Object Name, DisplayName, State`**:
   - Retrieves services that are currently running and set to start automatically, displaying their name, display name, and state. This helps in monitoring and managing system services.

CIM Queries

1. **`Get-CimInstance -ClassName win32_OperatingSystem | fl *:`**
   - Retrieves all properties of the Win32_OperatingSystem class using the CIM cmdlet and formats them in a list. This gives a complete view of all operating system properties.
2. **`Get-CimInstance -ClassName CIM_OperatingSystem | Select-Object Caption, Version, ServicePackMajorVersion`**:
   - Similar to the WMI version but uses CIM to get the same OS properties, which might have slight differences depending on the underlying implementation of the CIM vs. WMI providers.
3. **`Get-CimInstance -ClassName CIM_Processor | Select-Object Name, NumberOfCores, NumberOfLogicalProcessors`**:
   - Retrieves and displays selected processor properties using the CIM framework, which is the modern approach recommended over WMI.
4. **`Get-CimInstance -Query "SELECT * FROM CIM_OperatingSystem" | Select-Object Caption, Version, LastBootUpTime:`**
   - Executes a CIM query to retrieve all instances of the CIM_OperatingSystem class, selecting specific properties including the OS name, version, and the last boot-up time.
5. **`Get-CimInstance -ClassName Win32_Processor | Select-Object Name, LoadPercentage`**:
   - Retrieves information about each processor, including its name and current load percentage, using the CIM cmdlet. This is helpful for monitoring CPU usage in real-time.

Managing Active Directory

- **Active Directory Recycle Bin**
  - `Get-ADOptionalFeature 'Recycle Bin Feature'`
  - `Enable-ADOptionalFeature 'Recycle Bin Feature' -Scope ForestOrConfigurationSet -Target 'ohi.com'`
- **Forest and Domain info**
  - `Get-ADDomainController | fl *`
  - `Get-ADForest`
  - `Get-ADDomain`
- **Health Check**
  - `dcdiag /v /c /e`
  - `dcdiag /v /c /e /q #only show errors`
  - `dcdiag /test:replications`
  - `Dcdiagrepadmin /replsum`
  - `repadmin /showreps`
  - `dcdiag /test:dns`
  - `dcdiag /test:services`
  - `dcdiag /s:dc1 #Specify server`
- **Show which DCclient authenticate from**
  - `nltest /dsgetdc:abdelwahed.me`
- **Reset default domain policy**
  - `DCGPOFIX /target:dc`
  - `DCGPOFIX /target:domain`
  - `DCGPOFIX /target:bot`
- **Get all installed features**
  - `Get-WindowsFeature | where installed`

Managing Users

- **Get Details of a User:**
  `Get-ADUser -Identity "jdoe" -Properties *`
- **Create a New User:**
  `New-ADUser -Name "JaneDoe" -GivenName "Jane" -Surname "Doe" -SamAccountName "jdoe" -UserPrincipalName "jdoe@yourdomain.com" -Path "OU=Users,DC=yourdomain,DC=com" -AccountPassword (ConvertTo-SecureString "ComplexPassword123!" -AsPlainText -Force) -Enabled $true`
- **Modify a User Property:**
  `Set-ADUser -Identity "jdoe" -City "New York"`
- **Enable a Disabled User Account:**
  `Enable-ADAccount -Identity "jdoe"`
- **Disable an Active User Account:**
  `Disable-ADAccount -Identity "jdoe"`
- **Reset a User's Password:**
  `Set-ADAccountPassword -Identity "jdoe" -Reset -NewPassword (ConvertTo-SecureString -AsPlainText "NewComplexPassword!" -Force)`
- **Last Password Set**

```
Get-ADUser -Identity it01 -Properties * | select Name, PasswordLastSet
```

Managing Groups

- **Create a New Group:**

```
New-ADGroup -Name "ITSupport" -SamAccountName ITSupport -GroupCategory Security
-GroupScope Global -DisplayName "IT Support Team" -Path
"OU=Groups,DC=yourdomain,DC=com" -Description "This is the IT support team."
```

- **Add a User to a Group:**

```
Add-ADGroupMember -Identity "ITSupport" -Members "jdoe"
```

- **Get Members of a Group:**

```
Get-ADGroupMember -Identity "ITSupport"
```

- **Remove a User from a Group:**

```
Remove-ADGroupMember -Identity "ITSupport" -Members "jdoe" -Confirm:$false
```

- **Add all users starts with HR to group**

```
# Ensure $userDNs is always an array, even with one result
$userDNs = (Get-ADUser -Filter 'Name -like "HR*"')

# Add users to the group if any exist
if ($userDNs.Count -gt 0) {
  Add-ADGroupMember -Identity "sales_gr" -Members $userDNs
 Write-Output "Users added to the sales_gr group."
}
```

Managing Computers

- **Get Details of a Computer:**

```
Get-ADComputer -Identity "Computer01" -Properties *
```

- **Move a Computer to Another OU:**

```
Get-ADComputer -Identity "Computer01" | Move-ADObject -TargetPath
"OU=NewOU,DC=yourdomain,DC=com"
```

- **Disable a Computer Account:**

```
Disable-ADAccount -Identity "Computer01"
```

- **Enable a Disabled Computer Account:**

```
Enable-ADAccount -Identity "Computer01"
```

- **Move Computer to OU**

```
Get-ADComputer -Filter 'Name -like "c1*"' | Move-ADObject -TargetPath
"OU=pcs,DC=ohi,DC=com"
```

Managing OUs

- **Creating a New OU**

To create a new OU named "Marketing" in the domain root:

```
New-ADOrganizationalUnit -Name "Marketing" -Path "DC=yourdomain,DC=com"
```

If you want to create the OU in a specific location other than the domain root, adjust the -Path parameter accordingly.

- **Finding OUs**

To find OUs in your AD, use the Get-ADOrganizationalUnit cmdlet. For example, to find all OUs that contain the word "Sales":

```
Get-ADOrganizationalUnit -Filter 'Name -like "*Sales*"' | Select-Object Name,
DistinguishedName
```

This command lists the names and distinguished names (DNs) of the OUs.

Moving an OU

To move an OU to another OU, use the Move-ADObject cmdlet. For instance, moving the "Marketing" OU to under the "Departments" OU:

```
$targetOU = "OU=Departments,DC=yourdomain,DC=com"
$marketingOU = Get-ADOrganizationalUnit -Filter 'Name -eq "Marketing"' | Select-Object -ExpandProperty DistinguishedName
Move-ADObject -Identity $marketingOU -TargetPath $targetOU
```

This command first retrieves the DN of the "Marketing" OU and then moves it to the "Departments" OU.

- **Modifying an OU**

To modify properties of an OU, use the Set-ADOrganizationalUnit cmdlet. For example, to add a description to the "Marketing" OU:

```
Set-ADOrganizationalUnit -Identity "OU=Marketing,DC=yourdomain,DC=com" -Description "Marketing Department OU"
```

Removing an OU

To remove an OU, use the Remove-ADOrganizationalUnit cmdlet. This operation requires caution because it can result in the deletion of all objects within the OU. For example, to remove the "TemporaryAccounts" OU:

Remove-ADOrganizationalUnit -Identity "OU=TemporaryAccounts,DC=yourdomain,DC=com" -Recursive -Confirm:$false

- The -Recursive parameter allows the deletion of the OU and all its contents.
- The -Confirm:$false parameter suppresses the confirmation prompt. Be very cautious with this, especially when using -Recursive.

Protecting an OU from Accidental Deletion

You can set the "Protect from Accidental Deletion" flag on an OU with the Set-ADOrganizationalUnit cmdlet:

```
Set-ADOrganizationalUnit -Identity "OU=Marketing,DC=yourdomain,DC=com" -ProtectedFromAccidentalDeletion $true
```

This is an important setting to prevent unintentional removal of OUs and their contents.

Working with Event Logs in PowerShell

PowerShell provides several cmdlets for interacting with Windows Event Logs, enabling you to list, query, and manipulate event log data. Below are examples of how to perform common tasks related to event logs using PowerShell:

Listing Available Event Logs

- `Get-EventLog -List`

Retrieving Recent Entries from the System Log

- `Get-EventLog -LogName System -Newest 10`
- `Get-EventLog -LogName "system" -EntryType Error -Newest 10`

Finding Specific Events by Instance ID in the System Log

- `Get-EventLog -LogName Security -InstanceId 4625` #login failuar

Getting the Most Recent Error Entries from the Application Log

- `Get-EventLog -LogName Application -EntryType Error -Newest 5`

Filtering System Log Entries Related to Disk Issues

- `Get-EventLog -LogName System | Where-Object {$_.Message -like "*disk*"} | Select-Object -First 5`

Querying Security Log Entries Within a Specific Date Range

- `Get-EventLog -LogName Security -After "2023-01-01" -Before "2023-01-31"`

Using Get-WinEvent

Get-WinEvent is more versatile and powerful, particularly for querying the logs that relate directly to Active Directory like the Directory Service log or DNS server log. It supports complex queries through hash tables and XML, and it can handle the large event logs found on busy servers.

Querying Directory Service Log

For instance, to fetch events from the Directory Service log, you might use:

- `Get-WinEvent -LogName "Directory Service"`
- `Get-WinEvent -LogName "Security" -MaxEvents 50`
- `Get-EventLog -LogName Security | Where-Object {$_.EventID -eq 4624}`

Working with Hotfix Information

 # Find a specific hotfix installed on the system

- `get-hotfix | where-object {$_.HotfixID -eq "KB5034467"}`
- `Install-Module -Name PSWindowsUpdate`
- `Import-Module -Name PSWindowsUpdate`
- `get-wuhistory -MaxDate 2022-10-10`
- `get-wuhistory -Last 5`

 # Filter hotfixes by description and installation date

- `Get-HotFix | Where-Object {$_.description -like "*security*" -and $_.installedon -gt "01/01/2022"}`

Removing Hotfixes

If you need to uninstall a hotfix, you can do so provided the hotfix supports uninstallation. Not all hotfixes can be removed. For those that can, you might use wusa.exe or directly call the uninstaller if it's an .exe patch:

```
# Using wusa.exe for .msu files
```

- wusa.exe /uninstall /kb:HotfixID /quiet /norestart

Using variables

Variables in Windows PowerShell are essential for managing and manipulating data during scripting. They serve as containers that hold values or objects, which can be accessed and manipulated throughout the course of a script.

Functionality of Variables:

- **Data Storage:** Variables can store various types of data, such as log file names, user account details, or calculated dates to track user activity.
- **Handling Objects:** When storing objects, such as Active Directory user objects, a variable retains all properties of the object, allowing for detailed manipulation and retrieval of attributes.

Operations with Variables:

- **Viewing Variables:** You can view all stored variables using the `Get-ChildItem Variable:` or `Get-Variable` cmdlets.
- **Variable Naming:** Effective variable names are descriptive (e.g., `$user` for a user account) and should be clear and concise. PowerShell uses the dollar sign (`$`) as a prefix to denote variables.
- **Value Assignment:** Values are assigned using the `=` operator. Complex assignments might involve storing the results of commands directly into variables.
- **Displaying Values:** Use `Write-Host` to print variable values. The context (double vs. single quotes) affects whether the actual value or the variable name is displayed.
- **Clearing Variables:** To reset a variable's value, assign `$null` to it or use the `Clear-Variable` cmdlet.

```
$myVariable = "Data"
Clear-Variable -Name myVariable
$myVariable = $null
```

Variable Types:

- **Implicit Typing:** PowerShell typically infers the type of a variable based on the assigned value but can be explicitly defined to ensure accuracy (e.g., `[Int]$num2 = "5").`
- **Common Types:** Include strings, integers, floating-point numbers, date/time, and Booleans.
- **Type Confirmation:** Use the `.GetType()` method to verify the type of data stored in a variable.

Examples:

```
# Integer
$number = 25

# String
$name = "John"
$dir = dir "C:\MyFolder"
ls $dir
$dir | select Name, FullName, length
```

```
# Array
$colors = "red", "blue", "green"
$colors.GetType()
$colors.length
$firstColor = $colors[0]  # Red
$lastColor = $colors[-1]  # green, using negative index to access the last element
$colors[1]
$colors[1] = "black"
$colors += "Orange"
$colors
$colors[0..2]

# Hashtable
$names = @( "Aaron", "Albert", "Alphonse","Bernie", "Charlie", "Danny", "Ernie",
"Frank")
$names | where { $_ -like "A*" }
$names | ? { $_ -like "A*" }

----------------
```

interactive script

```
# Prompting the user for input and storing the input in variables
$user = Read-Host -Prompt "Write your full name"
$age = Read-Host -Prompt "Write your age"
$city = Read-Host -Prompt "Write your city"

# Creating a hashtable to store user information
$info = @{
  "Name" = $user
 "Age" = $age
"City" = $city
          }

# Pausing execution for 3 seconds
Start-Sleep -Seconds 2

# Displaying the information stored in the hashtable
Write-Host "Welcome $($info['Name']),"
Write-Host "Your Age is $($info['Age'])"
Write-Host "Your City is $($info['City'])"
```

- Storing a log file path: `$logFile = "C:\Logs\log.txt"`
- Deriving an email from a username: `$email = $username + "@example.com"`
- Calculating a past date: `$pastDate = (Get-Date).AddDays(-30)`

---

if, else and else if

```
$test = "test"
if ($test -eq "test")
{ Write-Host "if condition met" }
```

```
$test = "test"
 if ($test -eq "test2")
{ Write-Host "if condition met" }
elseif ($test -eq "test")
{ Write-Host "ifelse condition met" }
```

Foreach

```
$Names = @('Amy', 'Bob', 'Celine', 'David')
ForEach ($Name in $Names)
{
 Write-Host "Hi, my name is $Name!"
}
```

Functions

A PowerShell function is a named block of code designed to execute a specific task. When you invoke the function by its name, the code within its script block is executed. This setup is particularly useful for automating repetitive tasks, as it allows you to write the code once and reuse it multiple times by simply calling the function. In PowerShell, functions are defined with three main components: the **function** keyword, a function name, and a script block that contains the actual code, enclosed by curly brackets.

```
$user = Read-Host "Write Your Name"
$age = Read-Host "Write Your Age"

function Write-Greeting ($name, $age) {
    "Hello $name, you are $age years old."
}

Write-Greeting -name $user -age $age
```

Scripts

Example 1:

```
# Prompt the user to write the full path
$path = Read-Host -Prompt "Write the full path"
# Get file details in the specified path and calculate size properties
$get_files = Get-ChildItem -Path $path -File | Measure-Object -Property Length -Sum -
Maximum -Minimum -Average | Tee-Object -Variable datasize
# Store total size in bytes
$size_in_byte = $datasize.sum
# Convert total size to megabytes
$size_in_mb = $size_in_byte / 1MB -as [int]
# Convert total size to gigabytes
$size_in_gb = $size_in_byte / 1GB -as [int]
# Output the size information
Write-host "Total Size: $size_in_byte bytes" -BackgroundColor red -ForegroundColor Black
Write-host "Total Size: $size_in_mb MB" -BackgroundColor Green -ForegroundColor Black
Write-Host "Total Size: $size_in_gb GB" -BackgroundColor Yellow -ForegroundColor Black
```

Example 2:

```
$read_data = Get-Content -Path "D:\OneDrive - Abdelwahed\Desktop\DResources.txt"
foreach ($path in $read_data){
# Get file details in the specified path and calculate size properties
$get_files = Get-ChildItem -Path $path -Recurse | Measure-Object -Property Length -Sum -
Maximum -Minimum -Average | Tee-Object -Variable datasize
# Store total size in bytes
$size_in_byte = $datasize.sum
# Convert total size to megabytes
$size_in_mb = $size_in_byte / 1MB -as [int]
# Convert total size to gigabytes
$size_in_gb = $size_in_byte / 1GB -as [int]
# Get folder name from path
 $folderName = Split-Path -Path $path -Leaf
# Output the size information
Write-Host "Folder: $folderName" -BackgroundColor Blue -ForegroundColor White
Write-host "Total Size: $size_in_byte bytes" -BackgroundColor red -ForegroundColor Black
Write-host "Total Size: $size_in_mb MB" -BackgroundColor Green -ForegroundColor Black
Write-Host "Total Size: $size_in_gb GB" -BackgroundColor Yellow -ForegroundColor Black

}
```

Example 3:

```
# Load Windows Forms and configuration
Add-Type -AssemblyName System.Windows.Forms
# Create a new instance of FolderBrowserDialog
$folderBrowser = New-Object System.Windows.Forms.FolderBrowserDialog
$folderBrowser.Description = "Select a Folder"
# Show the FolderBrowserDialog
$result = $folderBrowser.ShowDialog()
# Check if the user clicked the 'OK' button
if ($result -eq [System.Windows.Forms.DialogResult]::OK) {
    $path = $folderBrowser.SelectedPath
    Write-Host "You selected: $path"
    # Proceed with file operations
    $get_files = Get-ChildItem -Path $path -Recurse | Measure-Object -Property Length -Sum
-Maximum -Minimum -Average | Tee-Object -Variable datasize
# Store total size in bytes
$size_in_byte = $datasize.sum
# Convert total size to megabytes
$size_in_mb = $size_in_byte / 1MB -as [int]
# Convert total size to gigabytes
$size_in_gb = $size_in_byte / 1GB -as [int]
# Get folder name from path
 $folderName = Split-Path -Path $path -Leaf
# Output the size information
Write-Host "Folder: $folderName" -BackgroundColor Blue -ForegroundColor White
Write-host "Total Size: $size_in_byte bytes" -BackgroundColor red -ForegroundColor Black
Write-host "Total Size: $size_in_mb MB" -BackgroundColor Green -ForegroundColor Black
Write-Host "Total Size: $size_in_gb GB" -BackgroundColor Yellow -ForegroundColor Black

}
```

Build Active Directory Hiercky

```
# Ensure the ActiveDirectory module is imported
Import-Module ActiveDirectory

# Create Organizational Units (OUs)
$ous = @("SRV", "PCs", "IT", "HR", "Sales", "Marketing", "QA")
foreach ($ou in $ous) {
    New-ADOrganizationalUnit -Name $ou -Path "DC=ohi,DC=com"
}

# Create Groups within specified OUs
$groups = @{
    "IT_Gr" = "OU=IT,DC=ohi,DC=com";
    "HR_Gr" = "OU=HR,DC=ohi,DC=com";
    "Sales_Gr" = "OU=Sales,DC=ohi,DC=com";
    "Marketing_Gr" = "OU=Marketing,DC=ohi,DC=com";
    "QA_Gr" = "OU=QA,DC=ohi,DC=com"
}

foreach ($group in $groups.Keys) {
    New-ADGroup -Name $group -GroupScope Global -Path $groups[$group]
}

# Create Users within specified OUs and Groups
$users = @{
    "IT" = 5;
    "HR" = 5;
    "Sales" = 5;
    "Marketing" = 5;
    "QA" = 5;
}

foreach ($ou in $users.Keys) {
    1..$users[$ou] | ForEach-Object {
        $userName = "$ou" + "0" + $_
        $userDN = "CN=$userName,OU=$ou,DC=ohi,DC=com"
        $pwd = ConvertTo-SecureString -AsPlainText "Learn@ohi" -Force
        New-ADUser -Name $userName -Path "OU=$ou,DC=ohi,DC=com" -AccountPassword $pwd -
Enabled $true

        # Add user to group
        $group = $ou + "_Gr"
        Add-ADGroupMember -Identity $group -Members $userDN
```

```
        }
    }

    # Assign all users in an OU to their respective group (if not already covered by creation
    loop)
    foreach ($group in $groups.Keys) {
        $ou = $group -replace "_Gr", ""
        $usersInOU = Get-ADUser -Filter * -SearchBase "OU=$ou,DC=ohi,DC=com"
        foreach ($user in $usersInOU) {
            Add-ADGroupMember -Identity $group -Members $user
        }
    }
```

1. Importing the ActiveDirectory Module

```
Import-Module ActiveDirectory
```

This line ensures that the Active Directory module is loaded into the PowerShell session. The Active Directory module provides cmdlets necessary for managing AD objects such as users, groups, and organizational units. It's important to ensure this module is imported before attempting any AD management tasks to avoid command errors.

2. Creating Organizational Units (OUs)

```
$ous = @("SRV", "PCs", "IT", "HR", "Sales", "Marketing", "QA")
foreach ($ou in $ous) {
    New-ADOrganizationalUnit -Name $ou -Path "DC=ohi,DC=com"
}
```

Here, an array of OU names is defined, and a foreach loop is used to iterate over each OU name and create it under the specified domain (DC=ohi,DC=com). New-ADOrganizationalUnit is the cmdlet used to create each OU.

3. Creating Groups within Specified OUs

```
$groups = @{
    "IT_Gr" = "OU=IT,DC=ohi,DC=com";
    "HR_Gr" = "OU=HR,DC=ohi,DC=com";
    "Sales_Gr" = "OU=Sales,DC=ohi,DC=com";
    "Marketing_Gr" = "OU=Marketing,DC=ohi,DC=com";
    "QA_Gr" = "OU=QA,DC=ohi,DC=com"
}

foreach ($group in $groups.Keys) {
    New-ADGroup -Name $group -GroupScope Global -Path $groups[$group]
}
```

This part defines a hashtable where each key-value pair consists of a group name and the path to the OU where the group should be created. The New-ADGroup cmdlet is used in a foreach loop to create each group in its respective OU with a global scope.

---

4. Creating Users within Specified OUs and Groups

```
$users = @{
    "IT" = 5;
    "HR" = 5;
    "Sales" = 5;
    "Marketing" = 5;
    "QA" = 5;
}

foreach ($ou in $users.Keys) {
    1..$users[$ou] | ForEach-Object {
        $userName = "$ou" + "0" + $_
        $userDN = "CN=$userName,OU=$ou,DC=ohi,DC=com"
        $pwd = ConvertTo-SecureString -AsPlainText "Learn@ohi" -Force
        New-ADUser -Name $userName -Path "OU=$ou,DC=ohi,DC=com" -AccountPassword $pwd -
Enabled $true

        # Add user to group
        $group = $ou + "_Gr"
        Add-ADGroupMember -Identity $group -Members $userDN
    }
}
```

This section creates users within each specified OU. A hashtable defines the number of users to be created for each OU. The users are created with sequential naming conventions and added to their respective groups. The password for each user is converted to a secure string and assigned.

5. Assign All Users in an OU to Their Respective Group

```
foreach ($group in $groups.Keys) {
    $ou = $group -replace "_Gr", ""
    $usersInOU = Get-ADUser -Filter * -SearchBase "OU=$ou,DC=ohi,DC=com"
    foreach ($user in $usersInOU) {
        Add-ADGroupMember -Identity $group -Members $user
    }
}
```

This final part ensures that any users already present in the OU are added to the corresponding group. It iterates through each group, retrieves all users in the OU, and adds them to the group.

Background Jobs and Scheduled Jobs

In PowerShell, you can use background jobs and scheduled jobs to run tasks asynchronously or at specific times, respectively. Both features are useful for automating tasks without interfering with the main workflow of your PowerShell session. Here's a brief overview of how to use both.

Background Jobs

Background jobs run a command or script in the background without interacting with the main session. You can continue working while the job runs. Here's how you can work with background jobs:

1. **Start a Background Job**: Use the Start-Job cmdlet to run a script block or command in the background.

- **Local Jobs**

```
$job1 = Start-Job -ScriptBlock { Get-Process }
$job2 = Start-Job -FilePath "C:\YourFolder\Script.ps1"  #Start a script as background job
Start-Job -ScriptBlock {Get-Process} -Credential (Get-Credential)  # Start job as a different user (Prompts for password)
```

Backing Up Files:

```
$job3 = Start-Job -ScriptBlock {
    Copy-Item -Path "C:\Users\Public\Documents" -Destination "\\BackupServer\BackupFolder" -Recurse -Force
}
```

Monitoring Disk Space:

```
$job4 = Start-Job -ScriptBlock {
    Get-PSDrive -PSProvider FileSystem | Select-Object Name, Used, Free | Export-Csv -Path "C:\DiskSpace.csv" -Append
}
```

Cleaning up temporary files

```
    $job5 = Start-Job -ScriptBlock {
    Get-ChildItem -Path $env:TEMP -Recurse | Remove-Item -Force -Recurse}
```

2. **Get Job Information**: Use Get-Job to see all running jobs and their statuses. For example:
```
Get-Job
```
3. **Receive Job Results**: Use Receive-Job to collect results from a completed job. For example:
```
Receive-Job -Job $job
Receive-Job -Job $job -Keep  #prevent the flush
$job | Wait-job | Receive-Job  #Waiting on a job to finish before getting the result
$job | Wait-job -Timeout 10  #Timeout a job if it runs too long (10 seconds in this example)
$job | Suspend-Job
$job | Resume-Job
```
4. **Remove a Job**: After collecting results, you might want to remove the job using Remove-Job. For example:
```
Remove-Job -Job $job
Get-Job | Where-Object { $_.State -eq 'Completed' } | Remove-Job
```

Scheduled Jobs

The `shell:startup` command in Windows is a shortcut to open the Startup folder. This folder contains shortcuts to applications that automatically start when a user logs into Windows. Managing the Startup folder is useful for both optimizing boot times and controlling what applications run automatically on system startup.

Scheduled jobs are used to run tasks at specific times or intervals. They are similar to cron jobs in Linux. Here's how you can set up scheduled jobs in PowerShell:

1. **Register a Scheduled Job**: Use the `Register-ScheduledJob` cmdlet to create a new scheduled job. You'll need to specify the script block and the trigger. For example:

   Delete Temp at startup

   ```
   Import-Module ScheduledTasks
   $trigger = New-ScheduledTaskTrigger -AtStartup
   $action = New-ScheduledTaskAction -Execute "Powershell.exe" -Argument "-NoProfile -
   Command `"Get-ChildItem -Path $env:TEMP -Recurse | Remove-Item -Force -Recurse`""
   Register-ScheduledTask -TaskName "ClearTempAtStartup" -Trigger $trigger -Action $action
   Get-ScheduledTask -TaskName "ClearTempAtStartup"
   ```

   Monthly User Account Review:

   ```
   $trigger = New-JobTrigger -Monthly -On 1 -At 6:00AM
   $scriptBlock = {
       Get-ADUser -Filter * -Properties LastLogonDate | Where-Object {
           $_.Enabled -eq $true -and $_.LastLogonDate -lt (Get-Date).AddMonths(-3)
       } | Export-Csv -Path "C:\Reports\InactiveUsers.csv"
   }
   Register-ScheduledJob -Name "MonthlyUserReview" -ScriptBlock $scriptBlock -Trigger
   $trigger
   ```

2. **List Scheduled Jobs**: You can list all scheduled jobs using:

   ```
   Get-ScheduledJob
   ```

3. **Get Job Results**: To see the results of executed scheduled jobs, use:

   ```
   $job = Get-ScheduledJob -Name "DailyProcessCheck"
   $job | Receive-Job
   ```

4. **Unregister a Scheduled Job**: If you need to cancel and remove a scheduled job, use:

   ```
   Unregister-ScheduledJob -Name "DailyProcessCheck"
   ```

Set Spooler service

```
# Define the service name using a variable
$serviceName = "Spooler"
# Get the current status and startup type of the specified service
$service = Get-Service -Name $serviceName
# Check if the startup type is not set to Automatic
if ($service.StartType -ne 'Automatic') {
    # Set the startup type to Automatic
    Set-Service -Name $serviceName -StartupType Automatic
    Write-Output "The startup type of the $serviceName service has been set to
Automatic."
}
```

```
# Check if the service is stopped
if ($service.Status -eq 'Stopped') {
    # Start the service
    Start-Service -Name $serviceName
    Write-Output "The $serviceName service was stopped and has now been started."
} else {
    # Output the current status if the service is not stopped
    Write-Output "The $serviceName service is already running."
}
```

Schedule job (script)

Every Day at 2 AM

```
# Define the action to start the PowerShell script
$action = New-ScheduledTaskAction -Execute 'Powershell.exe' `
  -Argument "-ExecutionPolicy Bypass -File C:\Scripts\CheckAndStartPrintSpooler.ps1"

# Define the trigger for daily at 2 AM
$trigger = New-ScheduledTaskTrigger -Daily -At 2AM

# Set the principal to run with highest privileges
$principal = New-ScheduledTaskPrincipal -UserId "SYSTEM" -LogonType ServiceAccount -
RunLevel Highest

# Settings to handle missed tasks
$settings = New-ScheduledTaskSettingsSet -StartWhenAvailable

# Create the scheduled task
Register-ScheduledTask -TaskName "Check and Start Print Spooler" -Action $action -
Trigger $trigger -Principal $principal -Settings $settings
```

Multiple Specific Times Each Day

```
# Define triggers for different times
$trigger1 = New-ScheduledTaskTrigger -Daily -At "10:00AM"
$trigger2 = New-ScheduledTaskTrigger -Daily -At "2:00PM"
$trigger3 = New-ScheduledTaskTrigger -Daily -At "4:00PM"
Create the scheduled task with all triggers
Register-ScheduledTask -TaskName "RunMultipleTimesDaily" -Action $action -Trigger
$trigger1, $trigger2, $trigger3 -Principal $principal
```

Every Few Hours Throughout the Day

```
# Define a trigger to start at 6 AM and repeat every 4 hours
$trigger = New-ScheduledTaskTrigger -Daily -At 6AM
$trigger.Repetition.Duration = 'P1D'  # Duration set to 1 day
$trigger.Repetition.Interval = 'PT4H' # Interval set to 4 hours
```

More Complex Schedules (e.g., Every 3 Hours and 30 Minutes)

```
# Define a trigger to start at 7 AM and repeat every 3 hours and 30 minutes
$trigger = New-ScheduledTaskTrigger -Daily -At 7AM
$trigger.Repetition.Duration = 'P1D'  # Duration set to 1 day
$trigger.Repetition.Interval = 'PT3H30M' # Interval set to 3 hours and 30 minutes
```

Administering Remote Computers

Before you can use PowerShell Remoting, it must be enabled on the target machines. This is typically done using the Enable-PSRemoting cmdlet, which configures the necessary WinRM listeners, starts the WinRM service, and sets up the appropriate firewall rules.

- **Example Command to Enable Remoting:**
  ```
  Enable-PSRemoting -Force
  Set-ExecutionPolicy RemoteSigned
  ```
- **Establishing a Remote Session**
  Once PowerShell Remoting is enabled, you can start a remote session. The Enter-PSSession cmdlet is used to start an interactive session with a remote machine.
- **Example Command to Start a Remote Session:**
  ```
  Enter-PSSession -ComputerName C19 -Credential $(Get-Credential)
  Get-PSSessionConfiguration
  ```

This command prompts you to enter credentials and then starts an interactive session with C19.

```
Exit-PSSession
```

Same

```
$client = New-PSSession –ComputerName C19
Enter-PSSession –Session $client
Exit-PSSession #(alias exit)
```

- **Executing Commands Remotely**

If you need to execute a single command or a script block on multiple remote computers, Invoke-Command is the tool for the job.

- **Command to Execute a Script Block Remotely:**

```
Invoke-Command -ComputerName Server01, Server02 -ScriptBlock {
    Get-Service | Where-Object {$_.Status -eq 'Running'}
}
```

or

```
Invoke-Command -ComputerName (Get-Content Names.txt)-ScriptBlock {
    Get-Service | Where-Object {$_.Status -eq 'Running'}
} #this command runs a script block on Server01 and Server02 that lists all running services.
Invoke-Command –Computer C19 –ScriptBlock { Get-Process –Name Note* } |
Stop-Process

Invoke-Command -Computer c19 - { Get-Process -Name Note*; get-service -name print* }
Invoke-Command { get-service -name wuau* }
```

Managing Remote Systems Using CIM

Both WMI and CIM can manage local and remote systems. For remote systems, CIM is preferred due to its better network transparency.

- **OS info**

```
Get-CimInstance -ClassName Win32_OperatingSystem -ComputerName c19 | select name
```

- **Restarting a Remote Computer**

```
        $remoteComputerName = 'C19'

# Retrieve the CIM instance of the Operating System from the remote computer
$os = Get-CimInstance -ClassName CIM_OperatingSystem -ComputerName $remoteComputerName

# Invoke the 'Reboot' method to reboot the remote computer
Invoke-CimMethod -InputObject $os -MethodName "Reboot"
```

- **Managing Services on Remote Computers**

```
$remoteComputerName = 'C19'
# Get the status of the Remote Desktop service
Get-CimInstance -ClassName Win32_Service -Filter "Name = 'TermService'" -ComputerName
$remoteComputerName |
    Select Name, State

# Start the service
$remoteComputerName = 'C19'
# Get the status of the Remote Desktop service
Get-CimInstance -ClassName Win32_Service -Filter "Name = 'TermService'" -ComputerName
$remoteComputerName | start-service
```

- **Monitoring CPU and Memory Usage**

```
$remoteComputerName = 'C19'
Get-CimInstance -ClassName Win32_Processor -ComputerName $remoteComputerName |
    Select-Object LoadPercentage  # CPU load
Get-CimInstance -ClassName Win32_OperatingSystem -ComputerName $remoteComputerName |
    Select-Object FreePhysicalMemory, TotalVisibleMemorySize  # Memory usage
```

- **Remote Jobs**

Restarting a service on multiple remote computers

```
    $computers = "DC1", "C19", "Server03"
    foreach ($computer in $computers) {
        $job = Invoke-Command -ComputerName $computer -ScriptBlock {
            Restart-Service -Name "Spooler"
        } -AsJob
    }
```

- **CIM Jobs**

Monitoring disk usage on remote systems

```
    $computers = "C19", "Dc1"
    foreach ($computer in $computers) {
        $job = Start-Job -ScriptBlock {
            param($comp)
            Get-CimInstance -ClassName CIM_LogicalDisk -ComputerName $comp | Select-
Object SystemName, DeviceID, @{Name="FreeSpaceGB"; Expression={$_.FreeSpace / 1GB -as
[int]}}
        } -ArgumentList $computer
    }
```

Monitoring Group Policy

1. Check Group Policy Application

To verify that Group Policy is applied correctly to a machine or user, you can use the Get-GPResultantSetOfPolicy cmdlet. This cmdlet generates a report that shows which Group Policies are applied and their settings.

```
# Generate RSoP report for the computer
Get-GPResultantSetOfPolicy -ReportType Html -Path "C:\GPReports\ComputerRsop.html" -Computer "c19"
# Generate RSoP report for a user
Get-GPResultantSetOfPolicy -ReportType Html -Path "C:\GPReports\UserRsop.html" -User "Domain\User"
```

2. Retrieve All GPOs in the Domain

You can list all Group Policy Objects in your domain using Get-GPO. This is helpful to quickly review the existing policies.

```
# Get all GPOs in the domain
Get-GPO -All
```

3. Get a Specific GPO

If you need details about a specific GPO, including its settings, you can retrieve it by name or ID:

```
# Get specific GPO by name
Get-GPO -Name "Default Domain Policy"
```

4. Monitor GPO Changes

To track changes to GPOs, you can use the Get-GPPermissions and Get-GPOReport cmdlets. You can run these commands periodically and compare the outputs to detect changes.

```
# Get permissions set on a GPO
Get-GPPermissions -Name "Default Domain Policy" -All
# Generate a report for a specific GPO
Get-GPOReport -Name "Default Domain Policy" -ReportType Xml -Path "C:\GPReports\DefaultDomainPolicy.xml"
```

5. Check for GPOs Linked to an OU

You can list all GPOs linked to a specific Organizational Unit (OU) using Get-GPInheritance.

```
# Get GPO inheritance for an OU
Get-GPInheritance -Target "OU=Sales,DC=Domain,DC=com"
```

6. Backup and Restore GPOs

It's a good practice to regularly back up GPOs. You can automate this process using PowerShell.

```
# Backup all GPOs
Backup-GPO -All -Path "C:\GPOBackup"
# Restore a GPO from backup
Restore-GPO -Name "Default Domain Policy" -Path "C:\GPOBackup"
```

7. Audit GPO Changes

To facilitate auditing, you can enable auditing for Group Policy changes via the Group Policy Management Console or using PowerShell to set the necessary audit policies. Then, use event logs to monitor changes.

```
# Check audit settings (example for Advanced Audit Policy Configuration)
Auditpol /get /category:"Object Access"
```

8. Scripting Regular Checks

To automate these checks, you can create a PowerShell script that runs at scheduled intervals using Task Scheduler. This script can check the health of GPOs, generate reports, and even alert you via email if it detects important changes.

Sharing and Permissions with PowerShell

```
Get-SmbShare
Get-SmbOpenFile
Get-SmbConnection
Get-SmbSession
Get-Acl "D:\Public_Share" | fl *
```

- **Controlling Bandwidth**

```
Install-WindowsFeature FS-SMBBW
Set-SmbBandwidthLimit -Category Default -BytesPerSecond 5242880
Set-SmbBandwidthLimit -Category LiveMigration -BytesPerSecond 20971520
Set-SmbBandwidthLimit -Category StorageReplication -BytesPerSecond 20971520
```

- **Stop the default admin share**

```
Set-ItemProperty -Path "HKLM:\SYSTEM\CurrentControlSet\Services\LanmanServer\Parameters"
-Name "AutoShareWks" -Value 0 -Type DWord
Restart-Service -Name LanmanServer
```

- **Create a new SMB share**

```
New-SmbShare -Name "PublicShare" -Path "D:\Public_Share" -FullAccess "DOMAIN\AdminGroup"
-ChangeAccess "DOMAIN\PowerUsers" -ReadAccess "Everyone"
```

```
$acl = Get-Acl "D:\Public_Share"
$acl | fl *
$acl.AccessToString #Access Summary
$acl.Access #Access Details
$acl | Get-Member
```

- **Define new access rules**

```
$rule1 = New-Object System.Security.AccessControl.FileSystemAccessRule("Administrators",
"FullControl", "ContainerInherit,ObjectInherit", "None", "Allow") #for local users and groups
```

```
$rule1 = New-Object
System.Security.AccessControl.FileSystemAccessRule("abdelwahed.me\IT_GR", "FullControl",
"ContainerInherit,ObjectInherit", "Allow")
```

```
$rule2 = New-Object System.Security.AccessControl.FileSystemAccessRule("DOMAIN\HR_GR",
"Modify", "ContainerInherit,ObjectInherit", "Allow")
```

```
$rule3 = New-Object System.Security.AccessControl.FileSystemAccessRule("Everyone",
"Read", "ContainerInherit,ObjectInherit", "Allow")
```

- **Add the rules to the ACL (in memory)**

```
$acl.SetAccessRule($rule1)
$acl.SetAccessRule($rule2)
$acl.SetAccessRule($rule3)
$acl.AccessToString
```

- **# Apply the ACL to the folder**

```
Set-Acl "D:\Public_Share" $acl
$acl.AccessToString
```

- **Change Ownership**

```
$newOwner = New-Object System.Security.Principal.NTAccount("DOMAIN",
"NewOwnerUsername")#to Domain user or group
$newOwner = New-Object System.Security.Principal.NTAccount("Administrators") #to local user
or group
$acl.SetOwner($newOwner)
Set-Acl "D:\Public_Share" $acl
```

- **Enable Auditing**

```
$auditRule = New-Object System.Security.AccessControl.FileSystemAuditRule("everyone",
"Read,Write", "None", "None", "Success,Failure")
$auditRule = New-Object System.Security.AccessControl.FileSystemAuditRule("everyone",
"Read,Write", "None", "None", "Success,Failure")
$acl.AddAuditRule($auditRule)
Set-Acl "D:\Public_Share" $acl
$acl.Audit | Format-List *
```

- **Map Network Drive**

```
New-PSDrive -Name Z -PSProvider FileSystem -Root \\Server\Share -Persist
```

- `Write-Output 'First' && Write-Output 'Second'`
- `Write-Output 'First' || Write-Output 'Second'` **#First command succeeds, so the second command is not executed.**

Compressing Files or Folders

To create an archive (ZIP file) of specific files or an entire folder, you can use the Compress-Archive cmdlet. Here's the basic syntax and an example:

Syntax:

- `Compress-Archive -Path <PathToFilesOrFolders> -DestinationPath <PathToOutputZipFile>`

Example:

 # Compress multiple specific files

- `Compress-Archive -Path "C:\Users\example\Documents\File1.txt", "C:\Users\example\Documents\File2.txt" -DestinationPath "C:\Archives\MyFiles.zip"`

 # Compress an entire folder

- `Compress-Archive -Path "C:\Users\example\Documents" -DestinationPath "C:\Archives\DocumentsBackup.zip"`

Extracting Files from an Archive

To extract files from a ZIP archive, use the Expand-Archive cmdlet. Here's how you can do it:

Syntax:

- `Expand-Archive -Path <PathToZipFile> -DestinationPath <PathToExtractFiles>`

Example:

- `Expand-Archive -Path "C:\Archives\DocumentsBackup.zip" -DestinationPath "C:\Users\example\ExtractedFiles"`

Additional Options

Both Compress-Archive and Expand-Archive cmdlets offer additional parameters to customize their behavior:

- **-Update** (only with Compress-Archive): Updates an existing archive by adding new files or replacing the existing files in the archive with newer versions.
- **-Force**: Overwrites an existing archive or extracted files without prompting the user.
- **-CompressionLevel** (only with Compress-Archive): Specifies the level of compression (Optimal, Fastest, NoCompression).

Example of using additional options:

Compress a folder with maximum compression

- `Compress-Archive -Path "C:\Users\example\Documents" -DestinationPath "C:\Archives\DocumentsBackup.zip" -CompressionLevel Optimal -Force`

 # Update an existing archive with new files

- `Compress-Archive -Path "C:\Users\example\NewDocuments" -DestinationPath "C:\Archives\DocumentsBackup.zip" -Update`
